REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MARCH 2009	Conference Paper Postprint	April 2007 – March 2009

4. TITLE AND SUBTITLE

TOWARDS A SECURE PROGRAMMING LANGUAGE: AN ACCESS CONTROL SYSTEM FOR COMMONLISP

5a. CONTRACT NUMBER
N/A

5b. GRANT NUMBER
FA8750-07-2-0032

5c. PROGRAM ELEMENT NUMBER
62702F

6. AUTHOR(S)

Howard Shrobe

5d. PROJECT NUMBER
NICE

5e. TASK NUMBER
00

5f. WORK UNIT NUMBER
06

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Massachusetts Institute of Technology
77 Massachusetts Avenue
Cambridge, MA 02139

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFRL/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
N/A

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TP-2010-4

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES
This work was funded in whole or in part by Department of the Air Force contract number FA8750-07-2-0032. Presented at the 2009 International Lisp Conference, Cambridge MA March 22-25, 2009.

14. ABSTRACT
Computer security is becoming an increasingly important problem. Although, the problem is often described as one of network security, the core of the problem is the vulnerability of computer hosts. There are many underlying causes of computer vulnerability, but most of them are traceable to an underlying failure of language systems to enforce the semantics of object identify, extent and type. Compounding this failing is the inability of most programming languages to express constraints on information flow and access that would limit the damage due to a penetration. In this paper, we present an access control system for Lisp-like languages that allows precise specification of which actors are allowed to perform what operations on which types of objects. Making these controls non-bypassable in a language as dynamic as Common-lisp is a serious challenge; we present techniques based on use of the Meta-Object Protocol (MOP) that achieve this goal; furthermore, we outline how hardware support can provide stronger guarantees within this framework.

15. SUBJECT TERMS
security, access control, meta-object protocol

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Lok K. Yan
U	U	U	UU	14	19b. TELEPHONE NUMBER (*Include area code*) N/A

Towards a Secure Programming Language

An Access Control System for CommonLisp *

Howard Shrobe

MIT Computer Science and Artificial Intelligence Laboatory

hes@csail.mit.edu

Abstract

Computer security is becoming an increasingly important problem. Although, the problem is often described as one of network security, the core of the problem is the vulnerability of computer *hosts*. There are many underlying causes of computer vulnerability, but most of them are traceable to an underlying failure of language systems to enforce the semantics of object identify, extent and type. Compounding this failing, is the inability of most programming languages to express constraints on information flow and access that would limit the damage due to a penetration. In this paper, we present an access control system for Lisp-like languages that allows precise specification of which actors are allowed to perform what operations on which types of objects. Making these controls non-bypassable in a language as dynamic as Common-lisp is a serious challenge; we present techniques based on use of the Meta-Object Protocol (MOP) that achieve this goal; furthermore, we outline how hardware support can provide stronger guarantees within this framework.

* This material is based on research sponsored by the Air Force Research Laboratory, Intelligence Advanced Research Projects Activity under agreement number FA8750-07-2-0032. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory,Intelligence Advanced Research Projects Activity or the U.S. Government.

Keywords security, access control, meta-object protocol

1. Software Insecurity Arises from The Lack of Enforced Semantics

All modern commercial operating systems are vulnerable. Recent reports have included hacking into the bank account of French President Sarkozy[1] and suspected Russian use of "botnets" during the war with Georgia.[2] These are merely highly visible examples of a much larger problem. Figure 1 shows the rate at which attacks have been growing (4) while (10) documents that, despite years of patching, the skill level required to launch an attack has been **decreasing** due to accumulated tool development and software engineering by the attacking community.

Figure 2 shows a catalog of vulnerabilities in the Firefox browser collected in a previous project.[3] From this, it is clear that nearly all of the vulnerabilities arise from a failure of the underlying hardware and software to enforce the semantics of object extent (*e.g.*, buffer overflows), identity (*e.g.*, storage management bugs that lead to "dangling pointers" causing two conceptually different objects to occupy the same space), and type (*e.g.*, faulty method dispatch caused by passing an integer to a routine expecting an object).

It has been obvious for some time that the systematic use of a type safe language (*e.g.*, Lisp, Java, ML) could remove many of the current vulnerabilities. Nevertheless, prudent design is based on "defense in depth" i.e. providing many independent reasons why an attacker

[1] `http://www.informationweek.com/news/security/attacks/showArticle.jhtml?articleID=211300006`

[2] `http://www.gsnmagazine.com/cms/features/news-analysis/1042.html`

[3] Conducted as part of the DARPA Application Communities program

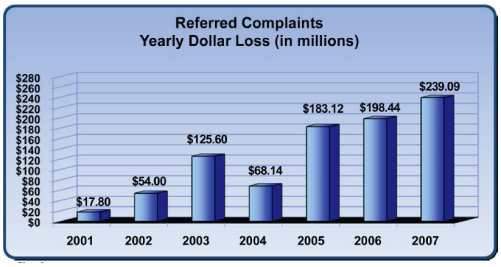

Figure 1. The Number of Successful Attacks is Growing Rapidly

Category	Description	N
Stack Overflow	Ill formed data causes overwriting of stack frame with data that is then branched to for execution	2
Array Access	Reference to data outside bounds of array which is then branched to for execution	2
Heap Overflow	Ill formed data causes overwriting of heap with data that is then branched to for execution	11
Dead Pointer	Use of an invalid pointer to inject data that is then branched to for execution	3
Trampoline Errors	Passing of invalid data to method dispatch routine causes branch to arbitrary position in memory	2
Garbage Collection	Violation of memory conventions causes garbage collector to create dead pointers	13

Figure 2. Vulnerabilities in Firefox

cannot succeed and providing mechanisms that limit the extent of the damage that a successful attacker can render. In addition, many popular types of applications, e.g. web servers, handle the data of many clients leading to a need for a principled mechanism for guaranteeing that the information flows allowed by the system are limited to those desired by its designers. Similarly, applications such as browsers, often support the use of dynamically loadable code from remote users (e.g. scripts), leading to a need to "sandbox" or compartmentalize the privileges available to such code.

Thus, in addition to the semantics of object identify, type and extent, it is crucial to define notions of locality (i.e. grouping of objects based on their shared context), privilege (i.e. the rights of individual computational actors to access and modify data) and information flow and to actively enforce these notions.

In a retrospective on the MULTICS (11) project, Saltzer and Schroeder (13) elicited several principles for the construction of such software:

1. *Complete mediation*: Every access to every object must be checked for authority– i.e. we must *both* (a) check the semantic validity of every operation and (b) check the authority on every instruction performed and every word of data.
2. *Least privilege*: Each module is granted only the minimal privileges necessary to do its job—this can be controlled to the level of individual privileged processor instructions on specific data types and words of memory.
3. *Separation of privilege*: Protection mechanisms should require that more than one condition should be met before access is permitted. More generally modules should distrust one another and check one another as in (5). This provides breach and error containment—rather than a single breach giving complete access to unrelated systems, this makes it necessary to compromise (or find errors in) a collection of components in order to subvert a system.

We have been working to create the mechanisms that can support these principles and have been designing language features that make the description of the mechanisms reasonably simple. As with the more basic properties of identity, type and extent, the critical step involves adding metadata to all objects within the system; in this case, the metadata specifies the "compartment" within which each object resides and the "principal" (i.e. a person or computational element of the software system) on whose behalf each thread executes. Coupled to this meta-data are a set of access rules, specifying which principals are allowed to perform what operations on which types of objects and an "access control system" that actively enforces these rules.

In this paper, we present the design of such an access control system for CommonLisp. CommonLisp is in many ways a natural vehicle for such a system, since it already enforces the more basic properties of object identity, type and extent without which any access control system could easily be subverted. In addition, the existence of powerful reflection capabilities eases the implementation. Finally, CommonLisp is a highly dynamic and open language, making it an interesting vehicle for exploring how completely one can enforce access controls in such an environment. On the other hand, CommonLisp is quite unsuited to the task because it lacks any non-bypassable isolation mecha-

nisms of its own (e.g. internal symbols of any package, including those that implement the substrate of the system, can be easily found). In the last section we discuss how to contain these problems and produce a secure framework.

2. The Model

Access control fundamentally is concerned with specifying who is allowed to do what to which objects. Conceptually, this can be expressed using an access control matrix (8; 9). The most common use of this model is the use of access control lists (ACL's) on files in commodity operating systems; these control the granting of a limited set of privileges (*e.g.* read, write, execute) to specific entities (*i.e.* users or groups) over coarse grained objects (*files, directories*). Such a model is at the least inconvenient and inexpressive. Later models (1) enrich the language by including notions such as hierarchies (or DAG's) of users, objects, and operations while (6; 2) introduce the notion of the roles a user might be playing and managing access in terms of such roles.

Our model draws on these, however, we are concerned with the systematic fine-grained control of access to *all* objects within the memory of the system, not just to external objects like files or directories. Thus, we center our model around CommonLisp objects and the generic-functions that operate on them. However, we extend this model as follows:

- **Objects**: Every object is an instance of a class. In addition, every object "lives" within a compartment.
- **Compartments**: A compartment is an aggregation of objects whose access rights are managed in common. Every object belongs to a single compartment. Compartments are themselves represented as objects and are therefore located within the class hierarchy.
- **Principals**: A principal is any active entity within the system, such as a user or a system components (*e.g* the scheduler). Principals are objects and therefore fit within the class system and live within a compartment. which
- **Threads**: Every thread has an associated principal on whose behalf the thread is executing. Every thread also has an associated compartment in which it allocates new objects.
- **Access Rules**: An access rule controls which principals are allowed to invoke which generic functions on objects in which compartments. An access rule is specified in a manner similar to a method, it contains:
  - The name of a generic function.
  - A class specifier for the principal.
  - A class specifier for the compartment of each argument.

  An access rule is applicable when:
  - The principal of the current thread is a member of the class specified by the principal class specifier.
  - The compartment of each actual argument is a member of the class specified by the compartment class specifier for the corresponding formal argument.

  The "body" of an access rule is limited to the keywords :permitted and :denied.

  Figure 3 shows an example access rule. This rule states that any Principal whose class is "Demo-Principals" can perform a Plus operation on any pair of operands both of which are in "Demo Compartments".

  When a generic function is applied to a set of arguments, the compartments of the arguments are used to fetch all applicable access rules. These are processed in most specific-first order, looking for an access rule whose body is :permitted or :denied. If an access rule with a :permitted body is found first, the thread is allowed to invoke the generic function on the arguments. If an access rule with a :denied body is found first, then an error is signaled. Finally, if there is no applicable access rule then an error is signaled. Thus, the default behavior is to deny access to any generic function (the default is to fault).
- **Gates**: A gate is a package of a generic-function, a compartment and a principal. A gate is a funcallable object; when it is called, the principal and compartment of the thread are rebound to those of the gate and the generic-function is called. The principal and compartment are rebound on return (normal or abnormal) from the generic function. These are very similar to the gates in MULTICS (11). Gates are the only means for changing a thread's compartment and principal.

Gates are objects and therefore live in a compartment. Access rules also control which principals can invoke which gates (based on the compartments of the gate and of the arguments).

We make the (invalid for CommonLisp) assumption that all operations are generic functions. As a result, the ability to allocate, access, and modify all objects is controlled by access rules.

In particular, one class of access rules control reader/write methods. These methods specify which principals can access which slots of objects in a particular compartment.[4]

A second class of access rules controls allocation of objects which is possible since the MOP specifies a series of generic-functions (*e.g.* allocate-instance, initialize-instance) that constitute the implementation of make-instance. First of all these rules control whether the principal of the running thread is attempting to allocate the object in the current compartment of that thread and secondly checks whether the principal is allowed to allocate in that compartment at all. Finally, these rules check that the principal has the right to initialize each of the slots as specified.

A third class of access rules specify which gates can be invoked by which principals.

Finally, access rules can be specified for arbitrary generic functions limiting a principal's ability to invoke specific services.

There are several consequences of the use of such access rules. Suppose an object is in compartment-1 and that principal-2 is not sanctioned to access the slots of objects in compartment-1. Then from the point of view of principal-2, object-1 is opaque; even if its slots contain objects that principal-2 can manipulate, principal-2 cannot discover these objects via object-1.

Another consequence is that privilege in the system need not be hierarchical; there need be no single actor (like the kernel or "root" user of an OS) that has all privileges. Each principal has a limited set of privileges, ideally as few as necessary to do its job. Inheritance can be used to compactly specify classes of principals that share privileges and classes of compartments whose privileges they share; but this need not imply strictly hierarchical layers of increasing privilege.

Information flows can be effected only by reading information from objects in one compartment and then writing this information into objects in another compartment. But, because, these actions are governed by access rules it should be possible, in principle, to formally analyze the information flows sanctioned.

Since the only way to change a thread's privileges is to invoke a gate and since gates can only be created if the access rules sanction the allocation and initialization operations involved, at least in principle it is possible to formally analyze what privileges are accessible to a thread.

Thus, as long as the set of access rules is static, predictable control of information flow is possible, even in a highly dynamic environment like CommonLisp.

3. An Application Example

In this section we describe a simple application that uses these building blocks. The application is a simplifed version of a graphical editor for military "couse of action diagrams" that we had built as part of a previous project. The tool is known as CCOAT (Commander's Couse of Action Tool). Our goal was to illustrate that we could simply retrofit access controls into this exiting body of code. In the retrofit, there are four compartments of data:

1. Top: Data in this compartment is privileged.
2. Blue: Data in this compartment represents the "blue view" of the situation.
3. Red: Data in this compartment represents the "red view" of the situation.
4. Common: Data in this compartment is open and accessible to all.

There are three types of users, each with different access rights:

1. Commanders: Are able to create, modify and access all application data.
2. Blue: Are able to create and modify data in the Blue compartment and read data in the Common compartment. Blue users cannot access or modify data in the Top or Red compartments. Blue users execute with the Blue compartment as their default compartment.

[4] In CommonLisp slots can also be accessed using the function slot-value. The MOP specifies that slot-value is defined in terms of the generic-function slot-value-using-class. Although we have not done so yet, the access rules for reader/writer methods should also control slot-value in exactly the same way.

```
(def-aif-method plus :permitter ((principal demo-principals)
 (p1 demo-compartments)
 (p2 demo-compartments))
  :permitted)
```

Figure 3. An Access Rule

3. Red: Are able to create and modify data in the Red compartment and read data in the Common compartment. Red users cannot access or modify data in the Top or Blue compartments. Red users execute with the Red compartment as their default compartment.

Figure 4 illustrate the code used to establish the compartments and principals. This involves nothing more than creating sub-classes of the base classes **User-Principals** and **User-Compartments**, and then instantiating instances of these classes. The class for the Common compartment has both the Red and Blue compartment as super-classes; this allows principals with access to Red compartments to inherit access to the Common compartment.

```
;;  Principals
(defclass ccoat-principals (user-principals) ())
(defclass ccoat-commander-principals (ccoat-principals) ())
(defclass ccoat-blue-principals (ccoat-principals) ())
(defclass ccoat-red-principals (ccoat-principals) ())

(defclass ccoat-compartments (user-compartments) ())
(defclass ccoat-commander-compartments (ccoat-compartments) ())
(defclass ccoat-blue-compartments (ccoat-compartments) ())
(defclass ccoat-red-compartments (ccoat-compartments) ())
(defclass ccoat-common-compartments
  (ccoat-red-compartments ccoat-blue-compartments)
   ())

(defparameter *ccoat-commander-compartment*
    (make-instance 'ccoat-commander-compartments
      :name "ccoat commander compartment"
      :magic-number (incf *user-compartment-number*)))
...
```

Figure 4. Code Establishing Compartments and Principals in CCOAT

Next we need to define our data structures and to specify which operations on these objects are available to which principals. Since the application is a graphical editor, most application object are instances of sub-classes of a base class called **point-like-objects**. Figure 5 shows the code necessary to extend the appropriate access rights to these objects for the three types of principals. To make the specification more compact, we created two macros **Extend-Multiple-Read-Permissions** and **Extend-Multiple-Write-Permissions**. We show the expansion of this macro for the Commander class but use the macros for the other classes.

Other data structures and access rules are specified in a similar manner. The CCOAT editor is implemented as a CLIM application. Separate CLIM application frames are set up for each user. The application provides commands to create objects representing different types of military units. Red users can only create red objects while blue users can only create blue object. Commanders can create objects in any compartment. The background map is created in the common compartment at application startup. All application objects are held in a common data structure that is shared by all users. The CLIM display loop in each application frame iterates over the objects in this common data structure, presenting each on its display. However, when a red user attempts to display a blue object (or a blue user attempts to display a red object) and access violation is signalled. This is because displaying the object requires accessing its **X** and **Y** slots and this is prohibitted by the access rules. The CLIM display loop catches and ignores this access violation and then goes on to display the next object. The net result is that the Red user's display shows only the background and red objects, which the blue user's display shows only the background blue objects. The commander's display shows all objects. Figure 6 shows an example of a set of CCOAT displays.

4. Using the Model to Build Secure Components

In the previous section we illustrated how the building blocks described in Section 2 can be used to build applications that segregate data into multiple compartments, extending different access rights to different classes of principals. In this section, we discuss how these building blocks allow one to build operating sys-

```
(defclass point-like-object ()
  ((x :initarg :x :accessor x)
   (y :initarg :y :accessor y)))

;;; commanders can see anything
(defmethod x :permitter
  ((principal ccoat-commander-principals)
   (point-like-object ccoat-compartments))
  t)
(defmethod y :permitter
  ((principal ccoat-commander-principals)
   (point-like-object ccoat-compartments))
  t)

;;; commanders can change anything
(extend-multiple-write-permissions (x y)
 ((new-value t)
  (principal ccoat-commander-principals)
  (point-like-object ccoat-compartments)))

;;; red can mung red
(extend-multiple-write-permissions  (x y)
 ((new-value t)
  (principal ccoat-red-principals)
  (point-like-object ccoat-red-compartments)))

;;; blue guys can see blue data
(extend-multiple-read-permissions  (x y)
 ((principal ccoat-blue-principals)
  (point-like-object ccoat-blue-compartments)))
;;; red guys can see red data
(extend-multiple-read-permissions (x y)
 ((principal ccoat-red-principals)
  (point-like-object ccoat-red-compartments)))

;;; blue can mung blue
(extend-multiple-write-permissions (x y)
 ((new-value t)
  (principal ccoat-blue-principals)
  (point-like-object ccoat-blue-compartments)))
```

Figure 5. CCOAT Access Rules

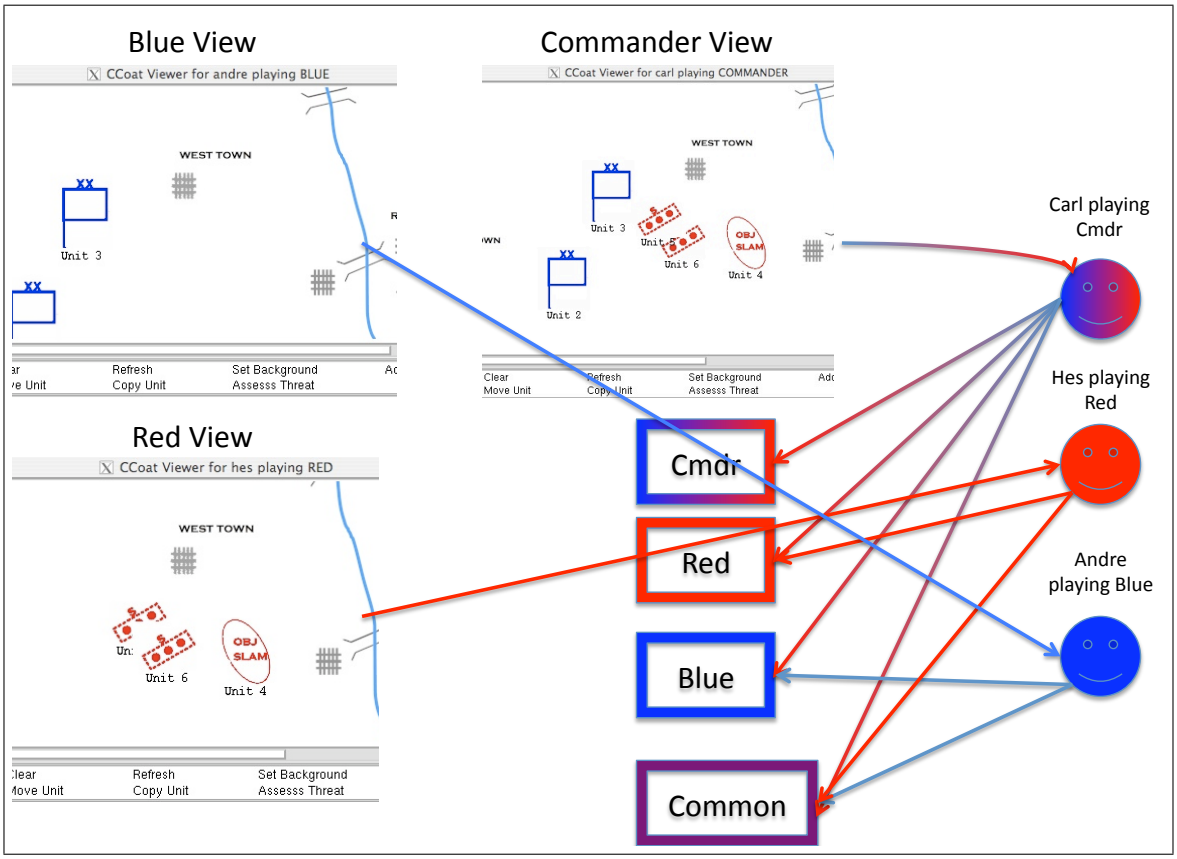

Figure 6. An Example CCOAT Display

tem software in a novel way that is mindful of the need to maintain security properties such as privacy and integrity. In section 1 we cited several key principles for such software were described by Saltzer and Schroeder

1. *Complete mediation*.
2. *Least privilege*.
3. *Separation of privilege*.

Current computer systems violate these principles with abandon. Operating systems, for example, divide the world into a "kernel" that runs with unlimited privileges and "user spaces" that are isolated from one another and the kernel. However, within each user space there are few tools for decomposing privilege and for guaranteeing that all operations are, in fact, checked for authority. Complicating the problem is that the kernel in most systems has grown to enormous size offering a large "attack surface" with the attraction for attackers of achieving full control of a system. Much server software, even those that run in user mode, present the same problem; a large complex body of software manages many users' data without adequate controls on access and information flow.

Compartments, principals, access rules and gates provide a set of building blocks for a different way of structuring such software. To illustrate the process, and the design patterns that emerge, we will use an example of a "log manager" a utility that accepts log entries from a variety of sources and commit these to persistent storage (presumably in encrypted form). The challenge is to build such a utility in such a way that there is a very low possibility that it will leak information between its various clients.

The starting point of the design is to consider who are the actors in the scenario, what data these actors manipulate, what interactions between the actors are required and what constraints exist on information flow. From these one then defines a set of principals corresponding to the actors and compartments into which the data is aggregated. In the current example, we create a principal for the Log-Manager *per se* and a principal for each client (*e.g.* User-1 and User-2); we also create a compartment for each of the actors (*e.g.* Log-Manager-Compartment, User-1-Compartment, User-2-Compartment). The access rules specify that the Log-Manager can access data in the Log-Manager-Compartment and that User-1 can access any data in Compartment-1 in any way desired (and similarly for User-2 and Compartment-2). At this point we have 3 isolated sub-systems completely incapable of interacting; it is as if we have 3 separate disconnected computers.

Of course, it is desired that User-1 and User-2 should be able to communicate with the Log-manager. However, it is not desired that log entries from User-1 (and data that these reference) should be able to be transmitted to User-2 via the Log-Manager. Instead of acting like completely separate computers, we'd like it to appear there is a FIFO connecting User-1 and the Log-Manager and a separate FIFO connecting User-2 and the Log-Manager and that the data in these FIFO's are read-only.

We can do this as follows: For each user, the Log-Manager creates an additional principal; this principal can be thought of as a proxy for the Log-Manager in its interactions with each user. Thus we have 2 new principals: Log-Manager-Acting-for-User-1 and Log-Manager-Acting-for-User-2. In addition, the Log-Manager create two new compartments (User-1-Log-Manager-Compartment, User-2-Log-Manager-Compartment) to support the interaction between the log manager and each of the users. As the owner of these compartments, the Log-Manager grants itself the right to create new gates in these compartments.

We next consider the significant operations in the interaction. These are 1) Create a new log-entry data structure and 2) Add a log-entry to the log data-structure, represented by the Create-Entry and Add-Entry generic functions. We also impose access rules that only allow Log-Manager-Acting-for-User-1 to call Create-Entry and Log-Manager-Acting-for-User-1 to allocate a log-entry in User-1-Log-Manager-Compartment (and similarly for User-2). We will refer to these compartments and principals as "satellites". Notice that we are using a defense in depth strategy: To build a new entry requires calling Create-Entry, but Create-Entry needs to call make-instance (and its subroutines); unless a principal is granted access to both generic-functions it cannot even build an entry.

Nevertheless, at this point the components are still isolated since Create-Entry can only be called by the Log-Manager-Acting-for-User-1 principal. In order to allow the thread acting on behalf of User-1 to actually build an entry, the Log-Manager must therefore create a Gate whose procedure is Create-Entry and whose

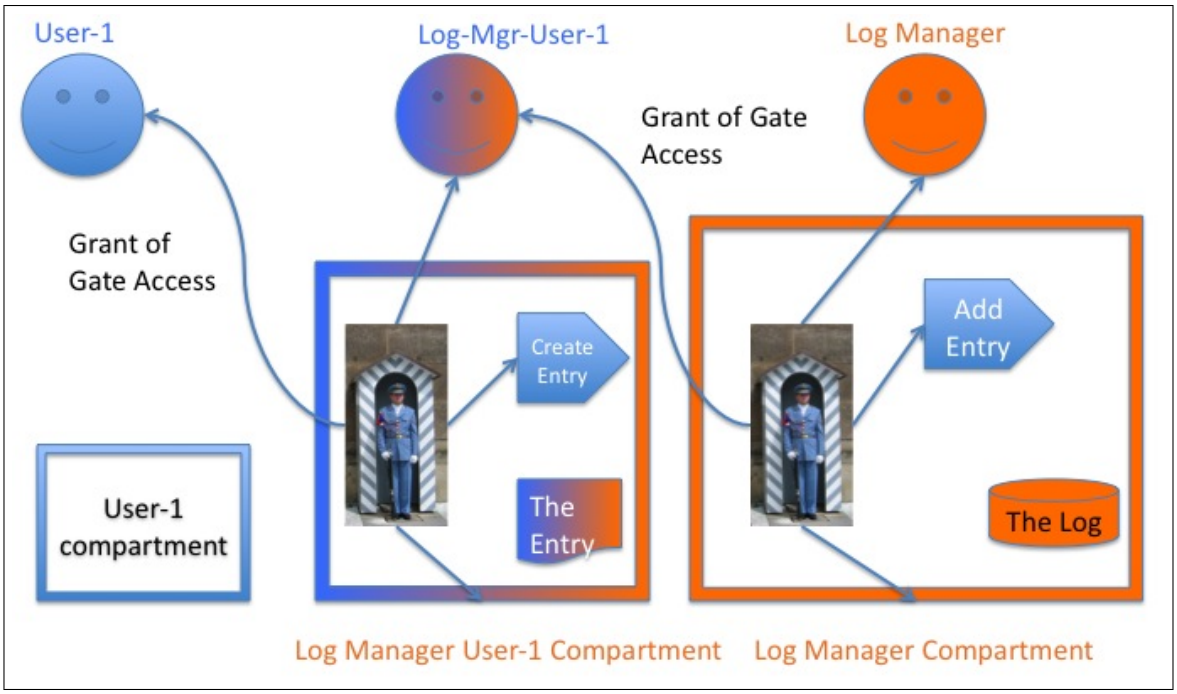

Figure 7. Pattern for limited interactions

principal and compartment are those belonging to Log-Manager-Acting-for-User-1; this gate is created in the Log-Manager-Acting-for-User-1 compartment. Finally, the Log-manager grants the User-1 principal the right to invoke gates in the Log-Manager-Acting-for-User-1 compartment. Given these structures, User-1 can invoke the gate, temporarily switching to the satellite principal and compartment; acting as the satellite principal the thread calls Create-Entry, allocating a new log entry in the satellite compartment. This is shown in figure 7. Notice in the figure that the Log-Manager's compartment also contains a gate whose procedure is Add-Entry, and whose compartment and principal are those of the Log-Manager. The Log-Manager grants the principal Log-Manager-Acting-for-User-1 the right to invoke this gate. After Create-Entry builds the new log-entry, it invokes this gate, switching to the principal and compartment of the Log-Manager itself, and adding the new entry to the log. At this point, both the first and second gates return, restoring the principal and compartment to that of User-1.

A bit of consideration shows that the structure we have created provides only the most limited information flows among the users and the Log-Manager. The only operation that User-1 can perform on its satellite compartment is to allocate new log entries and pass them on to the log manager. Neither User-1 nor the Log-Manager have write permission to their shared satellite compartment. Thus, the Log-Manager cannot act as a vehicle for leaking information from User-1 to User-2.

We refer to the pattern illustrate above as the "hub and satellite" pattern and it is a very common way of structuring interactions among the components of a software system where highly controlled information flows are desired. This pattern illustrates how we fulfill each of the 3 principles mentioned above:

1. **Complete Mediation**: All generic functions are controlled by access rules. Every operation is monitored.
2. **Least Privilege**: Each principal is granted the most limited privileges it needs to get it jobs done. For example, the Log-Manager can only add entries to its log; it cannot access information in users' compartments. Users similarly are never granted access to Log-Manager data; they are only allowed to create entries and pass them to the Log-Manager.
3. **Separation of Privilege**: In order to add an entry to the log, one must have permission both to call Create-Entry and to allocate objects in the satellite

compartment. Both checks would have to be bypassed by an attacker.

5. Implementation Techniques

As suggested in the previous section, the abstract model is implemented using the tools provided by CLOS and in particular by the MOP. The main techniques are as follows:

- All classes inherit from a common base class that provides a slot for the compartment of the object.
- Access rules are compiled (in an obvious manner) into methods whose qualifier is :permitter.
- We define a new method combination, that is essentially an OR method combination, except that it fetches :permitter methods.

  Corresponding to each original generic function, a new generic function is generated (with an uninterned mangled name) and that uses this method combination. Thus the template for each of these checking functions is:

```
(or
 (eql :permitted
      <list of :permitter-method calls>)
 (error ...  ))
```

- We define a new method combination, that is used by all generic functions. The combined method that is built first checks the access control rules and then calls the normal elements of the combined method. The template for each combined method is:

```
(apply <corresponding-gf>
       <the thread's principal>
       (mapcar #'compartment arguments))
<rest of normal method combination>
```

  It would have perhaps been more elegant to use the MOP to control how the :permitter methods are fetched, in particular to fetch them based on the types of the compartments of the arguments rather than the types of the arguments. The MOP provides an entry for doing this, compute-discriminating-function as well as lower level entry points, compute-applicable-methods and compute-applicable-methods-using-classes. In some implementations there is a cache of previously computed applicable (combined) methods and this is checked in compute-discriminating-function; if the applicable method is already in the cache, then there is no need to call compute-applicable-methods. However, we only want to change how each actual argument is used to fetch applicable methods, i.e. we want to provide the ability to substitute a "argument for dispatching" for each actual argument. This must, therefore, be done within compute-discriminating-function before checking the method cache. Were this not true, we would have only modified the lower level compute-applicable-method.

The current principal and the current compartment are represented as slots in a special object that represents the "machine state". Gates are implemented as instances of a subclass of funcallable-objects. When invoked the gate, "rebinds" the compartment and principal slots of the machine state, runs its code and then restores the machine state. Rebinding the machine state, is done by saving the machine state in internal slots of the gate and modifying the appropriate slots of the object representing the machine state and then reversing these steps on exit. Of course, the compartment and principal slots of the machine state should not be modifiable by any other method. This is achieved by wrapping the accessor methods for these slots with special, implementation dependent code, that checks who is calling the accessor and signaling an error unless the caller is a gate.

6. Dynamic Access Rules

We have so for, for the most part, been assuming that the set of compartments, principals, and access rules is static. However, in Section 4 we talked about creating such entities on the fly (*e.g.* we discussed the idea of the Log-Manager creating new satellite compartments and principals and access rules governing them). There are several reasons why the situation cannot be completely static, including:

- As illustrated in section 4 many components act like servers; as new clients enter the system (*e.g.* new users log in, new web sessions are opened, etc.) there is a need to create the appropriate infrastructure to serve their needs while preserving the desired inter-client isolation.

- It is imperative that there be mechanisms for revoking access rights that have been extended to bad players. Such revocation is inherently a dynamic operation.
- We would like the system to be dynamic in the sense that it should be possible to introduce new services, applications, etc. while the system is running. These will need to dynamically instantiate a set of principals, compartments and access rules as part of their startup transient.

However, we do not want this dynamism to become a back door for subverting existing controls. In particular, if anybody is allowed to add or remove a new access rule, then this could be used to deny legitimate services or to extend illegitimate privileges to some set of users. In effect, the protection system itself could easily become the locus of attack. We, therefore, need a way to allow dynamism but to impose constraint on that dynamism; we also need to ensure that these mechanisms are not bypassable.

To achieve these goals we extend the existing framework by noting that access rules are implemented as methods and that the MOP provides generic functions that implement the process of adding and removing methods, in particular: Add-Method and Remove-Method. Since these are generic functions, they can have access rules applied to them,[5] thereby limiting who can change generic which types of access rules.

Generic functions are objects and therefore fall into a class system. This allows us to make all generic functions in an application, for example, inherit from a single base-class. In addition, principals are also objects falling within the class hierarchy, so we can create Principal classes for the developers (and users) of the application. This allows us to compactly specify who is allowed to change the methods implementing the generic functions making up the application, by providing :permitter methods that use these base classes as method specializers.

During the development period of the application, it is convenient to extend blanket rights to all developers to change any generic functions that are part of the application. After deployment, however, we might want to change who has this right, limiting it to a subclass of developers empowered to patch the running system. Each of these can be easily and compactly specified.

At this stage, we have 1) A set of access rules governing who can perform which generic functions (these are implemented as :permitter methods on the application generic functions) and 2) A set of access rules governing who can change these generic functions (these are implemented as :permitter methods on Add-Method and Remove-Method).

The next step in the process is to create a set of access rules governing who can change the access rules. Like all access rules, these are implemented as :permitter methods. Furthermore, like the access rules that govern who can change generic functions, these are also methods on Add-Method (and Remove-Method). To see why, recall that Add-Method takes two arguments: A generic-function and the method to be added. In the normal case, the first argument is a generic-function implementing application functionality and our access rule limits who can add (remove) new methods to that generic function. However, if the generic-function is itself Add-Method (or Remove-Method), then adding a new :permitter method controls who is allowed to add (or remove) methods from the generic-function Add-Method (or Remove-Method). But the access rules governing who can change access rules are, in fact, :permitter methods for the Add-Method (Remove-Method) generic-function. Thus imposing the correct :permitter methods controls who can change access rules. The meta-circularity closes the loop as a final set of access rules controls who can add (or remove) :permitter methods to Add-method. This is shown in figure 8. In particular, this last :permitter is applicable to itself; once in place it says that it cannot be removed and that no other such method can replace or override it.

The consequence is that once the final :permitter method is in place it establishes a chain of non-bypassable protections. It protects itself and in turn it protects the access rules that limit who can change access rules. These in turn control who can change the methods that implement application generic functions and the :permitter methods that control who can invoke which application generic functions.

In addition to the mechanisms already discussed, it is necessary to develop conventions governing which

[5] In the current implementation this isn't strictly true, since we actually only control generic functions whose meta-class is a special class of our design. Add-method and remove-method are standard generic functions. We deal with this by using an :around method whose specializers are all T; this calls out to a generic function of the right meta-class.

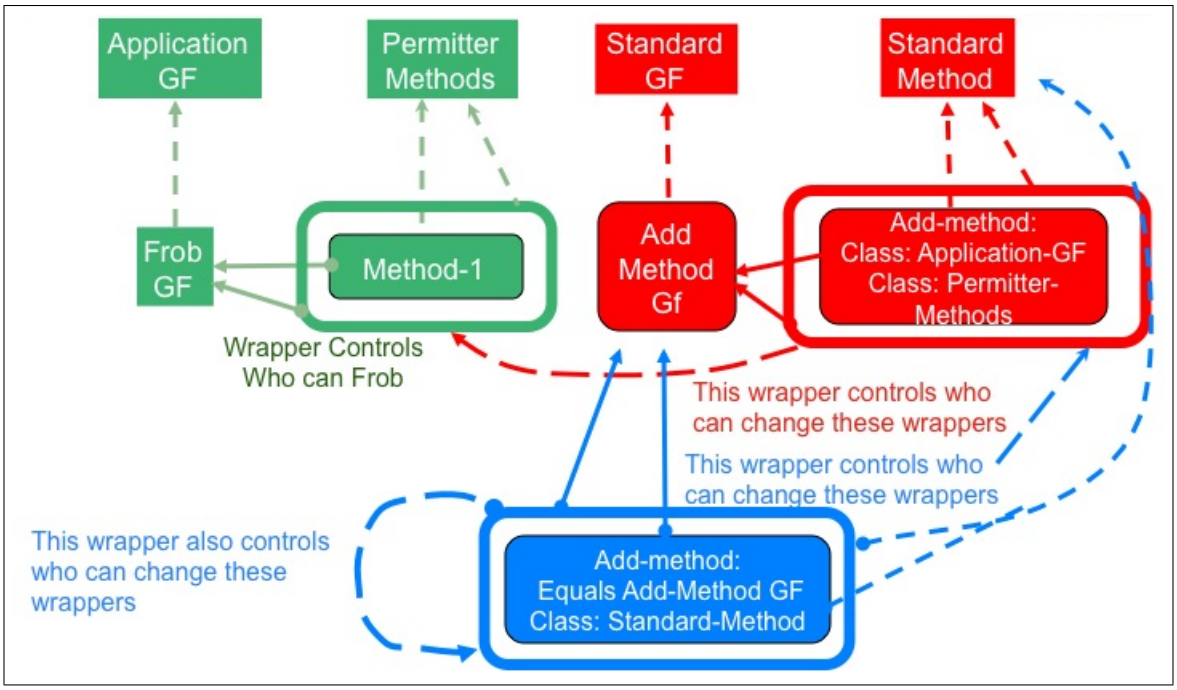

Figure 8. Controlling Dynamism

principals are allowed to impose what kinds of access rules. As we saw in Section 4 there is often a natural notion of a system component (*e.g.* the Log-Manger) owning a set of compartments, particularly compartments that it has created. Obviously, this component should have the unique right to specify access rules over data in these compartments. But can this component (*i.e.* the Principal representing the component *per se*) delegate this authority to other principals and if so, to which ones. This is still the subject of future development and requires further elaboration of our policy model.

7. CommonLisp Presents Challenges

In section 6, we showed that we can limit the degree of dynamism involved in changing access rules. But we have been ignoring rather extreme weaknesses of CommonLisp that allow the model to be subverted. These weaknesses include:

- Not all CommonLisp data are class instances operated on by generic functions. In particular, list structures are built from CONS cells which provide no slots to represent their compartment. Other immediate data (*e.g.* FIXNUMS) similarly have no easy way to represent compartments. This violates the basic assumption of our implementation that all data live in compartments and are only operated on by generic functions that can be "wrapped" with :permitter methods. It would be possible to associate a compartment with such structures using weak hashtables.[6] This works for everything but numeric data, where we might want to distinguish the integer 1 in compartment A from the integer 1 in compartment B. Unfortunately, all 1's are both equal and eq to one another so there is no way to distinguish them without boxing them into larger structures or to use hardware tagging as suggestion in Section 8.
- Normal user code can easily call internals of the language system implementation. The package system is the closest thing to a module system provided, but internal symbols of any package can be found and invoked by any code. Worse yet, the implementations normally provide reasonably good tools for discovering the internal functions in a package.
- Key internal data-structures of the CommonLisp language implementation may be built from such data and may be operated on by non-generic functions that cannot be wrapped. When combined with the previous point, it becomes reasonably straightforward to find and change the internal data struc-

[6] We thank one of the reviewers for this suggestion.

tures of method combination, method caching, etc. Indeed, generic-function-methods returns a list of methods that can be modified.

- Function cells of symbols can be accessed and overwritten; this is true for generic functions as well as normal functions. This means that an attacker can inject code that could, for example, change the function-cell of Add-method, bypassing our entire scheme.
- Any function, including a generic function, can be Advised.
- The slot-access protocol includes a very low level interface, standard-instance-access, that cannot be further specialized.[7] This can be used to bypass our higher level wrappers.

These are not insurmountable problems, but dealing with all of them would require much effort. Many of the problems enumerated above would go away if every object (including immediate data) were to have a compartment and if every function were a generic function. Under such conditions, the techniques illustrated in section 4 could be applied systematically to the entire language implementation.

8. Hardware Support

In (3) Knight and Brown describe a processor that can provide direct hardware support for the model described in this paper and we (Tom Knight, Andre deHon and I) are currently working on elaborating and implementing such a processor. The core idea is to structure the machine as a tagged processor inspired by and similar to the Lisp Machine (7) and other tagged processors (12). In our case, the tag encodes both the datatype and the compartment of the word it is attached to. Special internal processor registers hold the current principal and compartment and these are only accessible via special instructions.

In the hardware, access rules are applied to each instruction, checking the datatype and compartment of each operand. Read access rules check the compartment of the object being read from as well as the compartment of the datum fetched from the object. Write rules check the compartment of object being written into, the datum in the slot being overwritten and of the write datum. There is a procedure call instruction which checks the compartment of the invoked procedure. As a consequence it is possible to protect specific words of memory from being overwritten. Using these hardware level tools, it becomes possible to layer the implementation; control over slot readers and writers is translated into hardware level enforcement and is totally non-bypassable. Using this base, the mechanisms for wrapping all generic functions can also be made non-bypassable, meaning that the entire access control system would be inviolable.

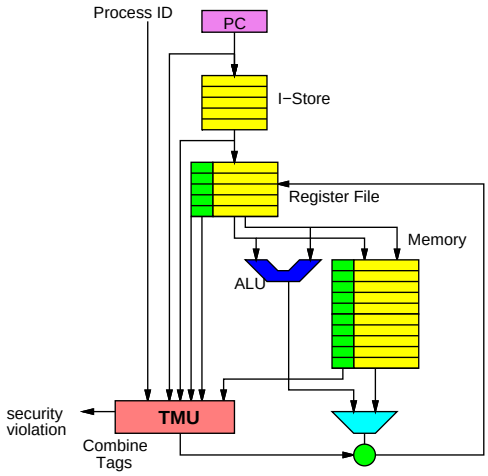

Figure 9. The Tag Management Unit

The hardware mechanisms proposed in (3) is surprisingly simple and are shown in figure 9. All words whether in memory or machine registers (including the program counter (PC)) have a tag including the datatype and the compartment of the datum. The Tag Management Unit (TMU in Figure 9) runs in parallel with the normal data path of the processor and is implemented as a modest size hashing cache, called a Hash Execution Unit or HASHEX; this is similar in structure to the TLB's used for virtual memory translation and should occupy about the same amount of chip space. The HASHEX computes a hash of the current instruction, the tags of all the operands, the current principal, and the tag of the PC. It uses this hash value to fetch a word from the HASHEX cache memory; this returns a flag indicating whether the operation is permitted and the tag of the result of the instruction. This is combined with the result of the normal processor data path and written back into the register file. The HASHEX is required to complete its execution by the time that the normal datapath commits its results; this seems to present no serious challenge. In addition, the HASHEX is a cache and it is possible that the cache can miss and

[7] We thank one of the reviewers for calling this to our attention.

need to be refilled from an access rule table in main memory. If the cache is not large enough to hold the working set of access rules, then the miss rate will be too high and will seriously degrade performance. We are still studying how big this needs to be, but preliminary results suggest that this too is manageable.

9. Summary

In this paper we have introduced an access control model for Lisp-like languages. The key elements of this model are compartments, principals, access rules and gates. We presented an illustration of how this model can be used to achieve several of the principles guiding secure system construction stated by Saltzer and Schroeder (least privilege, separation of privilege, complete mediation). We presented an implementation technique in which all objects are extended to include an extra piece of metadata, the compartment and in which access rules are compiled into wrappers methods that limit access to generic functions based on the compartments of the arguments and the current principal. In the last two sections we described how the extreme openness and dynamism of CommonLisp make it difficult to implement the model in a completely non-bypassable manner and how novel tagged hardware can address this problem.

10. Acknowledgements

I would like to thank Andre deHon, Thomas Knight and John Mallery who have collaborated with me in the larger project of which this effort is a part. Many of the ideas contained here were inspired by interactions with them.

References

[1] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. A domain and type enforcement unix prototype. In *SSYM'95: Proceedings of the 5th conference on USENIX UNIX Security Symposium*, pages 127–140, Berkeley, CA, USA, 1995. USENIX Association.

[2] John Barkley. Implementing role-based access control using object technology. In *First ACM Workshop on Role-Based Access Control*, November 30 – December 1 1995.

[3] Jeremy Brown and Jr. Thomas F. Knight. A minimal trusted computing base for dynamically secure information flow. Technical Report Aries Project Technical Report 15, MIT AI Lab, November 2001.

[4] Internet Crime Complaint Center. 2007 internet crime report. Technical report, The National White Collar Crime Center, Bureau of Justice Assistance, Federal Bureau of Investigation, 2007.

[5] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 184–194, 1987.

[6] David F. Ferraiolo and D. Richard Kuhn. Role-based access controls. In *Proceedings of the 15th NIST-NSA National Computer Security Conference*, October 13–16 1992.

[7] R.D. Greenblatt, T.F. Knight Jr., J. Holloway, D.A. Moon, and D.L. Weinreb. The lisp machine. In *Interactive Programming Enviornments*. McGraw-Hill, 1984.

[8] Butler W. Lampson. Protection. In *Fifth Princeton Symposium on Information Sciences and Systems*, pages 437–443, March 1971. (reprinted in Operating Systems Review, 8,1, January 1974, pp. 18 - 24).

[9] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973.

[10] Howard F. Lipson. Tracking and tracing cyber-attacks: Technical challenges and global policy issues. Technical Report CMU/SEI-2002-SR-009, CMU CERT, 2002.

[11] Elliot I. Organick. *The MULTICS system: An examination of its structure*. The MIT Press, 1972.

[12] Elliot I. Organick. *Computer System Organization: The B5700/B6700 Series*. Academic Press, 1973.

[13] Jerry H. Saltzer and Mike D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.